JavaScript - Ajax 3

Problema con las peticiones Ajax y cómo resolverlo:

Vamos a ver un ejemplo de una llamada a Ajax. Vamos a hacer una página que muestre en un párrafo el nombre y la descripción del producto del cual indiquemos su id en un input. En resumen, lo que hacemos es:

1. El usuario de nuestra aplicación introduce el código (id) del producto del que queremos ver sus datos.
2. Tenemos un escuchador para que al introducir un código de un usuario llamamos a una función *getProd()* que:
   - Se encarga de hacer la petición Ajax al servidor
   - Si se produce un error se encarga de informar al usuario de nuestra aplicación
3. Cuando se reciben los datos deben pintarse en la tabla

El archivo **productos.json** que contiene los datos:

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    }
 ]
}
```

La página html sería algo así:

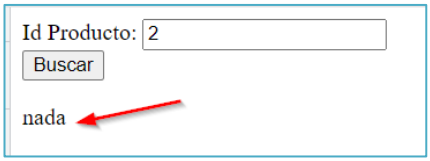
```
    <form id="addProduct">
         <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
         <br>
```

```
        <button type="submit">Buscar</button>
        <p id="p1">Aquí vendrán los datos</p>
    </form>
```

Si Ajax no fuera una petición asíncrona el código Javascript de todo esto sería algo como el siguiente (ATENCIÓN, este código **NO FUNCIONA**):

```
1   const SERVER = 'http://localhost:4000';
2
3   window.addEventListener('load', function() {
4       document.getElementById('addProduct').addEventListener('submit', (event) => {
5       event.preventDefault();
6       let idProd = document.getElementById('id-prod').value;
7       if (isNaN(idProd) || idProd == '') {
8         alert('Debes introducir un número');
9       } else {
10        const datos = getProd(idProd);
11        console.log(datos);
12        // pintamos los datos en la página
13        document.getElementById('p1').innerHTML = datos[0].name+"  "+datos[0].descrip;
14      }
15    })
16  })
17
18  function getProd(idProd) {
19    const peticion = new XMLHttpRequest();
20    peticion.open('GET', SERVER + '/productos?id=' + idProd);
21    peticion.send();
22    peticion.addEventListener('load', function() {
23      if (peticion.status === 200) {
24        const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
25        console.log(datos);
26        return datos
27      } else {
28        console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
29      }
30    })
31    peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
32  }
```

En la página se observa que aunque hagamos click varias veces, el contenido párrafo no se actualiza:



Por otro lado, viendo el terminal, observamos que la primera vez que hacemos click con el id=1, devuelve un código 200 (ha sido correcto el envío) y las siguientes veces devuelve código 304 (la información está en la caché porque es la misma petición y no hubo cambios en esos datos desde que devolvió código 200).

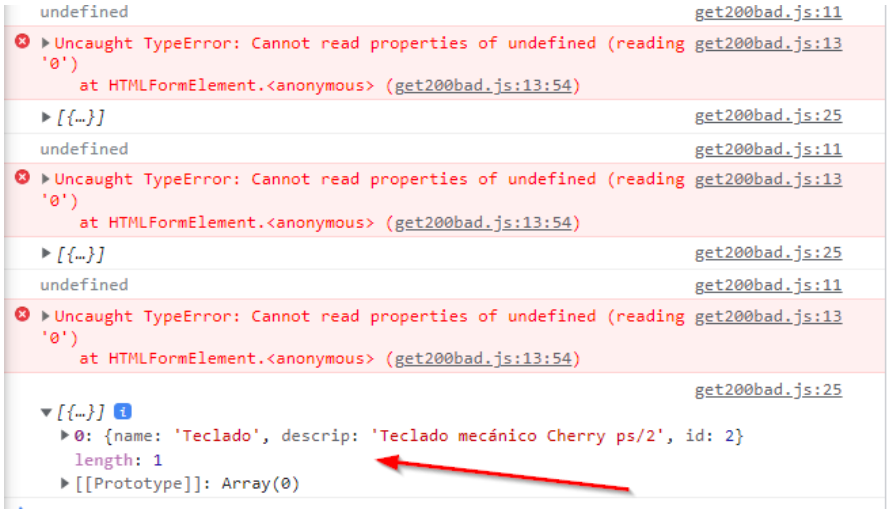
```
  Loading .\productos.json
  Done

  Resources
  http://localhost:4000/productos

  Home
  http://localhost:4000

  Type s + enter at any time to create a snapshot of the database
GET /productos?id=2 200 11.331 ms - 65
GET /productos?id=2 304 24.557 ms - -
GET /productos?id=2 304 37.382 ms - -
```

Si miramos en la consola, se aprecia que en la línea 25 los datos obtenidos son los mismos, al *parsear* se obtiene un array de longitud 1, cuyo único elemento es el objeto que buscamos.



Pero también se observa que la línea 11 recibe *undefined,* y más tarde, aparece el array que vemos en la línea 25. Lo que explica el error de la línea 13: intenta pintar los datos que aún no ha recibido porque la respuesta tarda en llegar. Este error debemos corregirlo.

Visto de otro modo: El array datos es undefined en la línea 11 porque cuando se llama a getProd(idProd), esta función no devuelve nada de inmediato, sino que devuelve tiempo después, cuando el servidor contesta, pero entonces nadie está escuchando.

La solución es que todo el código, no sólo de la petición Ajax sino también el de qué hacer con los datos cuando llegan, se encuentre en la función que pide los datos al servidor.

Está resuelto en el apartado siguiente:

Formulario para obtener (GET) un producto dado su id:

Este ejemplo resuelve el problema del apartado anterior.

```
    <form id="addProduct">
         <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
         <br>
        <button type="submit">Buscar</button>
        <p id="p1">Aquí vendrán los datos</p>
    </form>
```

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
    document.getElementById('addProduct').addEventListener('submit', (event) => {
      event.preventDefault(); // Cancela la acción predeterminada del evento 'submit'
      const idProd = document.getElementById('id-prod').value;
      if (isNaN(idProd) || idProd == '') {
        alert('Debes introducir un número');
      } else {
        getProducto(idProd);
```

```
      }
    })
})

function getProducto(idProd) {
  const peticion = new XMLHttpRequest();
  peticion.open('GET', SERVER + '/productos?id=' + idProd);
  peticion.send();
  peticion.addEventListener('load', function() {
    if (peticion.status === 200) {
      const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
      document.getElementById('p1').innerHTML = datos[0].name+"  "+datos[0].descrip;
      console.log(datos);
    } else {
      console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
    }
  })
  peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
}
```

Nota: Al poner **event.preventDefault()** en la función asociada a un evento, se cancela la acción predeterminada que tuviera ese evento.

Para pintar los datos, hay que tener en cuenta que GET devuelve un array (en este caso con un único elemento).

Este código funciona correctamente, pero tiene una pega: tenemos que tratar los datos (en este caso pintarlos en el párrafo) en la función que gestiona la petición, porque es la que sabe cuándo están disponibles esos datos. Por tanto, nuestro código es poco claro.

Esto se podría mejorar usando una función ***callback***.

Funciones callback

La idea es que creamos una función que procese los datos (`renderProd`) y se la pasamos a `getProd` como una función **callback** para que la llame cuando tenga los datos:

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
    document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      getProd(idProd, renderProd)
    }
  })
})

function renderProd(datos) {
    // aquí pintamos los datos. Habrá casos que será muy extenso.
    document.getElementById('p1').innerHTML = datos[0].name+"  "+datos[0].descrip;
```

```
}

function getProd(idProd, callback) {
  const peticion = new XMLHttpRequest()
  peticion.open('GET', SERVER + '/productos?id=' + idProd);
  peticion.send()
  peticion.addEventListener('load', function() {
    if (peticion.status === 200) {
      callback(JSON.parse(peticion.responseText));
    } else {
      console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición")
    }
  })
  peticion.addEventListener('error', () => console.error('Error en la petición HTTP'))
}
```

Hemos creado una función que se ocupa de renderizar (pintar) los datos y se la pasamos a la función que gestiona la petición para que la llame cuando los datos están disponibles (cuando petición.status===200).

Utilizando la función *callback* hemos conseguido que *getProd()* se encargue sólo de obtener los datos y cuando los tenga se los pasa a la función encargada de pintarlos en el documento de la página manipulando el DOM.

Promesas

Sin embargo hay una forma más limpia de resolver una función asíncrona, y es que el código se parezca al primero que hicimos que no funcionaba, donde la función `getProd()` sólo debía ocuparse de obtener los datos y devolverlos a quien se los pidió:

```
...
let idProd = document.getElementById('id-prod').value;
if (isNaN(idProd) || idProd == '') {
  alert('Debes introducir un número');
} else {
  const datos = getProd(iPprod);
  // y aquí usamos los datos recibidos para pintar los datos
}
...
```

La nueva forma es convirtiendo a `getProd()` en una ***promesa***.

Cuando se **realiza una llamada** a una promesa, quien la llama puede usar métodos que NO SE EJECUTARÁN hasta que la promesa se haya resuelto, es decir, hasta que el servidor haya contestado. Estos métodos son:

- **.then(*function(datos) { … }*)**: se ejecuta si la promesa se ha resuelto satisfactoriamente. Su parámetro es una **función.** Esta función recibirá como parámetro los datos que haya devuelto la promesa (que serán los datos pedidos al servidor). En este caso la promesa se los envía con ***resolve***(…)
- **.catch(*function(datos) { … }*)**: se ejecuta si se ha rechazado la promesa (normalmente porque se ha recibido una respuesta errónea del servidor). Ejecuta una función que recibe como parámetro la información pasada por la promesa al ser rechazada (que será información sobre el error producido). En este caso la promesa se los envía con ***reject***(…)

De esta manera el código quedaría:

```
let idProd = document.getElementById('id-prod').value;
if (isNaN(idProd) || idProd == '') {
  alert('Debes introducir un número');
} else {
  getProd(idProd)
      // en el .then() estará el código a ejecutar cuando tengamos los datos
    .then((datos) => {
      document.getElementById('p1').innerHTML = datos[0].name+"  "+datos[0].descrip;
    })
      // en el .catch() está el tratamiento de errores
    .catch((error) => console.error(error))
}
```

Para convertir a *getProd()* en una promesa solo hay que "envolver" las instrucciones de la función en una promesa.

```
function getProd(idProd){
    return new Promise((resolve, reject)=>{
        // Aquí el contenido de GetProd()
    }
}
```

Esto hace que devuelva un objeto de tipo *Promise* (`return new Promise()`) cuyo único parámetro es una función que recibe 2 parámetros:

- ***resolve***: función *callback* a la que se llamará cuando se resuelva la promesa satisfactoriamente.
- ***reject***: función *callback* a la que se llamará si se resuelve la promesa con errores.

El funcionamiento es:

- cuando la promesa se resuelva satisfactoriamente *getProd* llama a la función ***resolve()*** y le pasa los datos recibidos por el servidor. Esto hace que se ejecute el método ***.then*** de la llamada a la promesa que recibirá como parámetro esos datos. Se ejecuta .then() después de resolve()
- si se produce algún error se rechaza la promesa llamando a la función ***reject()*** y pasando como parámetro la información del fallo producido y esto hará que se ejecute el ***.catch*** en la función que llamó a la promesa. Se ejecuta .catch() después de resolve().

Por tanto, nuestra función *getProd* ahora quedará así:

```
function getPosts(idProd) {
  return new Promise((resolve, reject) => {
    const peticion = new XMLHttpRequest();
    peticion.open('GET', SERVER + '/productos?id=' + idProd);
    peticion.send();
    peticion.addEventListener('load', () => {
      if (peticion.status === 200) {
        resolve(JSON.parse(peticion.responseText));
      } else {
        reject("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
      }
    })
    peticion.addEventListener('error', () => reject('Error en la petición HTTP'));
  })
```

```
}
```

Se observa que el único cambio es la primera línea donde se convierte la función **getProd**() en una **promesa**, y que luego para "devolver" los datos a quien llama a *getProd* en lugar de hacer un *return*, que ya se ha visto que no funciona, se hace un *resolve* si todo ha ido bien o un *reject* si ha fallado.

Desde donde llamamos a la promesa **nos suscribimos a ella** usando los métodos ***.then()*** y ***.catch()*** vistos anteriormente.

Básicamente, lo que nos va a proporcionar el uso de promesas es un código más claro y mantenible, ya que el código a ejecutar cuando se obtengan los datos asíncronamente estará donde se piden esos datos y no en una función escuchadora o en una función *callback*.

Utilizando promesas vamos a conseguir que la función que pide los datos sea quien los obtiene y los trate o quien informa si hay un error.

El código del ejemplo de obtener un producto desde su id usando promesas sería el siguiente:

El código HTML sería igual que antes:

```
    <form id="addProduct">
         <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
         <br>
        <button type="submit">Buscar</button>
        <p id="p1">Aquí vendrán los datos</p>
    </form>
```

Y el código Javascript sería:

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
  document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      getProd(idProd)
        .then(function(datos) {
          // aquí pintamos los datos. Habrá casos que será muy extenso.
          document.getElementById('p1').innerHTML = datos[0].name+"  "+datos[0].descrip;
        })
        .catch(function(error) {
          console.error(error)
        })
    }
  })
})

function getProd(idProd) {
  return new Promise(function(resolve, reject) {
    let peticion = new XMLHttpRequest()
```

```
    peticion.open('GET', SERVER + '/productos?id=' + idProd);
    peticion.send()
    peticion.addEventListener('load', () => {
      if (peticion.status === 200) {
        resolve(JSON.parse(peticion.responseText))
      } else {
        reject("Error " + this.status + " (" + this.statusText + ") en la petición")
      }
    })
    peticion.addEventListener('error', () => reject('Error en la petición HTTP'))
  })
}
```

Nota: Los errores del servidor SIEMPRE llegan a la consola. En el ejemplo anterior si se produce un error de servidor aparecerá 2 veces en la consola: la primera que es el error original y la segunda donde está pintado con `console.error()`.

En este ejemplo, aunque pidamos un id de producto no existente en el servidor, devuelve un código 200 y un array vacío. El error que se ve en consola es porque no puede pintar un array vacío. Este error es capturado por ctach(), y permite que la página siga funcionando.

Más sobre Promesas en MDN web docs y en los documentos anexos.

Fetch

La API Fetch es una interfaz para manipular la información del servidor en forma de promesas.

Como el código que hay que escribir para hacer una petición Ajax es largo y repetitivo, la API-Fetch permite realizar una petición Ajax genérica que directamente devuelve en forma de **promesa**.

Básicamente lo que hace es encapsular en una función todo el código que se repite siempre en una petición AJAX (crear la petición, hacer el *open*, el *send*, escuchar los eventos, …).

La función *fetch* es similar a la función *getProd* que hemos creado antes, pero es genérica para que sirva para cualquier petición pasándole la URL. Su código es algo similar a:

```
function fetch(url) {
  return new Promise((resolve, reject) => {
    const peticion = new XMLHttpRequest();
    peticion.open('GET', url);
    peticion.send();
    peticion.addEventListener('load', () => {
        resolve(peticion.responseText);
    })
    peticion.addEventListener('error', () => reject('Network Error'));
  })
}
```

Hay 2 cosas que cambian respecto a nuestra función *getProd()*:

1. *fetch* devuelve los datos "en crudo" por lo que si la respuesta está en formato JSON habrá con convertirlos. Para ello dispone del método ***.json()*** que funciona como el `JSON.parse()`. Este método devuelve una nueva promesa a la que nos suscribimos con un nuevo *.then*. Es lo que se llama promesas encadenadas. Ejemplo.:

```
fetch('http://localhost:4000/productos?id=' + idProd)
  .then(response => response.json())    // los datos son una cadena JSON
  .then(myData => {      // ya tenemos los datos en _myData_ como un objeto o array
  // Aquí procesamos los datos (en nuestro ejemplo los pintaríamos en el párrafo de la página)
  console.log(myData)
  })
  .catch(err => console.error(err));
```

2. *fetch* llama a *resolve* siempre que el servidor conteste, sin comprobar si la respuesta es de éxito (200, 201, 304,…) o de error (4xx, 5xx). Por tanto, siempre se ejecutará el *then* excepto si se trata de un error de red y el servidor no responde (no ha habido respuesta).

Propiedades y métodos de la respuesta

La respuesta devuelta por *fetch()* tiene las siguientes propiedades y métodos:

- **status**: el código de estado devuelto por el servidor (200, 404, …)
- **statusText**: el texto correspondiente a ese código (Ok, Not found, …)
- **ok**: booleano que vale *true* si el status está entre 200 y 299 y *false* en caso contrario
- **json()**: devuelve una promesa que se resolverá con los datos de la respuesta convertidos a un objeto (a los datos les realiza un *JSON.parse()*)
- otros métodos para convertir los datos según el formato que tengan: **text()**, **blob()**, **formData()**, … Todos devuelven una promesa con los datos de distintos formatos convertidos.

El ejemplo que hemos visto con las promesas, usando *fetch* quedaría:

El código HTML sería igual que antes:

```
    <form id="addProduct">
         <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
         <br>
        <button type="submit">Buscar</button>
        <p id="p1">Aquí vendrán los datos</p>
    </form>
```

Y el código Javascript sería:

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', () => {
  document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      fetch(SERVER + '/productos?id=' + idProd)
        .then((response) => response.json())
        .then((datos) => {
           // aquí pintamos los datos. Habrá casos que será muy extenso.
           document.getElementById('p1').innerHTML = datos[0].name+"  "+datos[0].descrip;
```

```
        })
        .catch((error) => console.error(error))
    }
  })
})
```

Pero este ejemplo fallaría si hubiéramos puesto mal la *url* ya que contestaría con un 404 y fallaría al ejecutar el *then* intentando pintar un producto que no tenemos.

Gestión de errores con fetch

Según MDN, la promesa devuelta por la *API fetch* sólo es rechazada en el caso de un error de red, es decir, el *.catch* sólo saltará si no hemos recibido respuesta del servidor; en caso contrario la promesa siempre es resuelta.

Por tanto, para saber si se ha resuelto satisfactoriamente o no, debemos comprobar la propiedad ***.ok*** de la respuesta. El código correcto del ejemplo anterior gestionando los posibles errores del servidor sería:

```
// tratando los errores en fecth
const SERVER = 'http://localhost:4000';

window.addEventListener("load", () => {
  document.getElementById("addProduct").addEventListener("submit", (event) => {
    event.preventDefault();
    let idProd = document.getElementById("id-prod").value;
    if (isNaN(idProd) || idProd.trim() == "") {
      alert("Debes introducir un número");
    } else {
      fetch(SERVER + "/productos?id=" + idProd)
        .then((response) => {
          if (!response.ok) {
            // lanzamos un error que interceptará el .catch()
            throw `Error ${response.status} de la BBDD: ${response.statusText}`;
          }
          return response.json(); // devolvemos la promesa que hará el JSON.parse
        })
        .then((datos) => {
          // ya tenemos los datos formateados
          // Aquí procesamos los datos (en nuestro ejemplo los pintaríamos en la página)
          document.getElementById("p1").innerHTML =
            datos[0].name + "  " + datos[0].descrip;
          console.log(datos);
        })
        .catch((error) => console.error(error));
    }
  });
});
```

En este caso, si la respuesta del servidor no es *ok* lanzamos un error que es interceptado por nuestro propio *catch*

Otros métodos de petición POST, PUT…

Los ejemplos anteriores hacen peticiones GET al servidor. Para peticiones que no sean GET la función *fetch()* admite un segundo parámetro: un objeto con la información a enviar en la petición HTTP. Ej.:

```
// Otros métodos de petición POST, PUT…
fetch(url, {
    method: 'POST', // o 'PUT', 'GET', 'DELETE'
    body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
    headers:{
      'Content-Type': 'application/json'
    }
  }).then
```

Ejemplo de una petición para añadir datos:

```
fetch(url, {
    method: 'POST',
    body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
    headers:{
      'Content-Type': 'application/json'
    }
  })
  .then(response => {
    if (!response.ok) {
      throw `Error ${response.status} de la BBDD: ${response.statusText}`
    }
    return response.json()
  })
  .then(datos => {
    alert('Datos recibidos')
    console.log(datos)
  })
  .catch(err => {
    alert('Error en la petición HTTP: '+err.message);
  })
```

Más ejemplos en [MDN web docs]

Realiza una página web para que cualquier usuario de la página pueda dar de alta nuevos productos en *productos.json* utilizando fetch. El usuario introducirá el nombre y la descripición del producto, la API asignará el id que corresponda.

En la solución del ejercicio, si añadimos al objeto producto que se va a envía con POST la propiedad id con un valor puesto a mano, puede ocurrir:

- Si el número de id se repite, provocará un error 500, y el producto no se añadirá a la BBDD.
- Si el id no existe, se creará el producto con el valor id que se haya puesto a mano.
- Si no se pone id, la API colocará el número siguiente al último id del fichero. (Cuidado!!)

Nota para mí: Si en "Visual Studio Code" dejo abierta la carpeta desde la que he abierto (lanzado) la página con "Live Server" al hacer clic en el botón "submit", borra los inputs y el contenido de la consola, es como si reiniciase la página. Por consiguiente los datos no son persistentes, no se pueden pintar los datos en el DOM .

Hay varias soluciones:

1. Utilizando Live Server:
   - Abrir la aplicación con Live Server y dejar en ejecución esta página
   - Abrir otra carpeta (cambiar el folder) en VSC
   - Volver a la carpeta de la página ejecutada para poder ver los archivos fuente
   - Levantar el servicio json-server con el archivo json en cuestión
2. Usar un servidor web como apache, ya sea en el equipo local u otro equipo remoto